Center For Research in Embedded Systems and Technologi
Technical Report

Data Preteching Using Offline Learning

Jinwoo Kim and Krishna V. Palem
Georgia Institute of Technology
Atlanta, Georgia

Weng Fai Wong
National University of Singapore
Singapore

CREST-TR-01-005
GIT-CC-01-17
June 2001

DATA PREFETCHING USING OFF-LINE LEARNING

Jinwoo Kim[*], Krishna V. Palem[*], and Weng Fai Wong[+]

[*]Center for Research on Embedded Systems and Technology,
Georgia Institute of Technology
jinwoo@cc.gatech.edu

[*]Center for Research on Embedded Systems and Technology,
Georgia Institute of Technology
palem@ece.gatech.edu

[+]Dept. of Computer Science
National University of Singapore
wongwf@comp.nus.edu.sg

Abstract

An important technique for alleviating the memory bottleneck is data prefetching. Data prefetching solutions ranging from insertion of prefetch instructions by means of program analysis to strictly hardware prefetch mechanisms have been proposed. The former, however, is less successful for pointer intensive applications. In this paper, we propose a hardware solution that utilizes *off-line* learning algorithms. In essence, a sample trace of the application is fed into various off-line learning schemes. The results from these schemes are then loaded into a prefetching hardware at the appropriate point in the execution of the application to drive the prefetching. We propose a general architecture and scheme for such a process and report on the results of some of the experiments we performed.

1. Introduction

It is a well-established fact that as processor speed increases, memory becomes a serious performance bottleneck. While the introduction of caches significantly alleviated the problem, caching alone will not bridge the increasing performance gap between multi-issue processors running at very high clock speeds and memory. Data prefetching has been proposed as an additional tool to bridge this gap. Almost all state-of-the-art processors have some kind of data prefetching instructions. However, it is not easy to utilize these instructions effectively as they usually come with significant overheads. These include additional instruction processing, increased register pressure, the impact of additional instructions in the instruction cache, and the inability to adapt to dynamically changing demand for data. Due to these difficulties, various forms of hardware prefetching techniques have also been proposed. Of particular interest to us are techniques targeted at pointer intensive applications since these are significantly less amenable to the kind of program analysis necessary for software prefetching methods.

Existing hardware prefetching techniques require the prefetching hardware to perform some form of learning and prediction in real-time. This may necessitate a significant investment in hardware, or there may be an impact on the critical path of instruction processing. In the worst case, it can be both. In this paper, we propose a new prefetching architecture and a paradigm for using such an architecture that utilizes extensive profiling and powerful *off-line* learning algorithms.

In Section 2, we will describe some representative previous works on this subject. In Section 3, we will discuss the use of off-line learning algorithms. Our proposed architecture will be presented in Section 4 together with three learning algorithms that we tested. This is followed by experimental setup, results and a conclusion.

2. Previous Work

Research on memory hierarchy optimization can be classified into three broad categories: software approaches, hardware approaches and hybrid approaches. We will briefly mention some representative work that is relevant to our discussion. We refer the interested reader to a detailed survey on the matter that was recently published [26].

In the field of software prefetching early work include that done by Callahan, Kennedy, and Porterfield [2], and Klaiber and Levy [15]. The former proposed the insertion of data prefetch instructions in data intensive loops while the latter studied efficient architectural support mechanisms for data prefetch instructions. Mowry, Lam and Gupta [20] showed that careful analysis and selective prefetching could provide significant performance improvements in programs with regular nested loops. Lipasti *et. al.* [17] proposed a compile time heuristic called *Speculatively Prefetching Anticipated Interprocedural Dereference* (SPAID), for inserting prefetches into the instruction stream to reduce both the cost and the frequency of a certain class of data cache misses. Still other approach is that of Ozawa *et. al.* [22] in which they used cache miss heuristics to identify problematic loads and then to prefetch them. Luk and Mowry [18] introduced a method by which the compiler can insert software prefetch instructions for recursive data structures.

Perhaps aware of the potential difficulties of using software prefetching, there is significantly more research on the alternative hardware approach to data prefetching. One seminal work is Jouppi's proposal [12] of adding "stream buffers" to prefetch sequentially in conventional caches, there has been numerous suggestions for hardware prefetching. Prefetch strategies for vector and scalar processors were studied by Fu and Patel [8, 9]. Chen and Baer [4] proposed a lookahead data prefetching mechanism that combined stride information and instruction lookahead.

They also investigated a mechanism [5], known as the *Reference Prediction Table* (RPT), for prefetching data references characterized by regular strides. The RPT is a cache tagged with the addresses of load instructions. For each load instruction, the cache stores the previous memory address accessed by that instruction, the offset of that address from the previous load and flags to track of the data access patterns in a RPT. In this method, prefetches can be generated one iteration ahead of actual use but the problem was that memory latency hiding is dependent upon the execution time of a single loop iteration. Mehrota [19] proposed a hardware data prefetching scheme that attempts to recognize and use recurrent relations that exist in address computation of link list traversals. Extending the idea of correlation prefetchers [3], Joseph and Grunwald [11] implemented a simple Markov model to dynamically prefetch address references.

Hybrid approaches attempt to overcome drawbacks of pure software and hardware approaches by combining both. Karlsson, Dahlgren, and Stenstrom [13] proposed both a pure software version and a combination of software and hardware prefetching technique called “prefetch arrays” which can prefetch even short sequences linked data structure as the lists found in hash tables and trees where the traversal path is not known a priori. VanderWiel and Lilja [25] proposed a *data prefetch controller* (DPC), which combines low instruction overhead with the flexibility and accuracy of a compiler-directed prefetch mechanism.

3. Off-line Learning

Hardware predictors operate in two phases – a *learning* phase and a *prediction* phase. In the learning phase, the prediction facility is trained. Typically, this involves the updating of a prediction table or automaton. In the prediction phase, the learned table or

automaton is used to make prefetch requests. In some schemes, during the prediction phase, the prediction table or automaton may also be updated, i.e. the learning and prediction phases are interleaved.

A major drawback of existing hardware schemes is the need to perform learning and prediction both at run time. This severely limits the type of learning schemes that one can use. We propose overcoming this limitation by taking the learning phase off-line. By using sample traces collected from an application, prediction tables and automata can be trained off-line. This rests on the important assumption that the sample trace used for the training do correctly reflect the behavior of the application during its actual run. The success of hardware prefetch mechanisms, all of which are based on learning past patterns to predict future references, provides strong circumstantial evidence for this.

The factors determining the success of a prefetch scheme are *accuracy*, *timeliness*, *overhead* and *coverage*. Accuracy refers to the percentage of prefetch requests issued are actually used. An accurately predicted prefetch request is useless if it is issued too early or too late relative to the actual use of the data. Any prefetch mechanism will have an associated overhead (which may be in the form of additional instructions, hardware investment, or increased bus utilization) that must not be too significant. Finally, the scheme must be able to cover most of the loads. Unlike on-line schemes, off-line schemes can consider a significantly larger window of the sample trace and/or use more complex analysis and learning algorithms. This generally improves the accuracy of the prediction. Furthermore, by staying focus on program hotspots, coverage is improved. The issue of timeliness and overhead will be discussed when we outline our architectural solution.

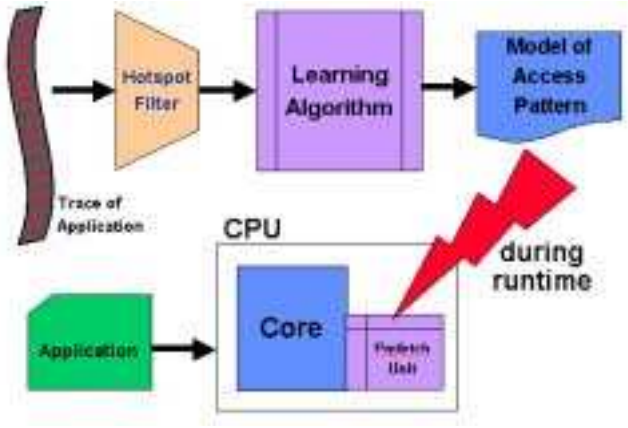


Fig. 1. Proposed Setup for Off-line Learning

4. The Proposed Architecture and System Setup

Fig. 1 shows the general structure of our proposed system. Sample traces of the application of interest are collected. In our experiments, these traces are first processed through a cache simulator so that we obtained only the miss traces. During the sample trace collection phase, the application is also profiled to identify the "hotspot" – sections of code that are frequently executed. Sections of the miss trace corresponding to a particular hotspot form the training sequences for that hotspot. These training sequences are then fed to a learning/analysis algorithm that outputs a prediction model for a particular hotspot. The prediction model is essentially a table with entries $(x, y_1, y_2, \mathrm{K}, y_n)$ where upon encountering miss address x, prefetch requests are issued for address y_1, y_2, …, y_n. The prediction models for each of the hotspot of an application are consolidated in a setup file. When the application is executed, prior to the entry into the hotspot, the prediction table is loaded into the prefetch hardware. The entry into a

hotspot is confirmed by means of checking the program counter. Once in a hotspot, the prediction hardware with the loaded prediction model for the hotspot is turned on. When a cache miss occurs, the prediction hardware simply checks the miss address against its table. If it is present, it will issue the corresponding prefetch requests. Otherwise, it does nothing. Note that it is possible to use either L1 or L2 data cache misses to drive the predictor. Furthermore, we assume a separate prefetch buffer [12] that is distinct from the L1 data cache in which prefetched data is stored. This is to prevent unnecessary pollution of the L1 data cache.

Timeliness in our proposed architecture depends on three factors – whether the prediction table can be loaded sufficiently ahead of time, whether there is sufficient distance between prefetch requests and the actual uses of data, and whether they is sufficient bandwidth to handle the prefetch requests. Since the prediction tables are pre-calculated, it should be easy to preload the tables at a sufficiently early point of time. The second factor will depend on the analysis/learning algorithm. The last factor will depend on the memory architecture and the frequency of prefetch request. At this moment, we are unable to estimate the hardware investment needed to build the proposed architecture. To get a fairer picture, we have compared our scheme against that of using significantly bigger caches. We further believe that the autonomy of the prefetch unit implies a minimal impact on the critical path of instruction processing in the core processor. We shall now describe the three learning/analysis algorithms we used in our experiments. We wish to emphasize the generality of our proposed solution and that one can implement many other off-line learning algorithms.

4.1 Simple Markov Predictor

This simple predictor is similar to the one used by Joseph and Grunwald [11]. Let **T** be the sample miss trace of an application. For two miss addresses, x, $y \in$ **T** say, the probability $P(y \mid x)$, i.e. the probability of x being followed immediately by miss address y, is computed. For each $x \in$ **T** in the miss trace, we compute $N(x) = \{P(y \mid x)\}$ where , $x, y \in$ **T** and $y \neq x$. In addition, from the trace we compute $f(x)$ which is the frequency of occurrences of x in **T**.

Next, we fix the size of the prediction table. Since in practice, this will not accommodate all miss addresses, we need a hashing algorithm to access the table. Let $h(x)$ be the hashing function that maps x to its entry in the prediction table. We used a lookup mechanism that is similar to cache tag checking. This ensures that the prediction table can be checked very quickly. We are now ready to construct the prediction table. We iterate through the rows of the prediction table. Let k be a row in the prediction table. From the set $\{x \mid h(x) = k, x \in$ **T** $\}$, we select a miss address x with the highest $f(x)$. In other words, of all the miss addresses that map to the same row, we pick the one with the highest frequency of occurrences in the sample trace. Let p be the number of prefetch request entry per row. Having selected x, we simply use the p miss addresses of $N(x)$ with the highest probabilities. For our experiments, we chose p to be 4.

4.2 Windowed Markov Predictor

This is similar to the simple Markov predictor except that in the computation for $N(x)$, instead of considering only the miss addresses that immediately follows x, we use a window of size w and consider all miss addresses within the window. In other words, if y_1, y_2, …, is the sequence of miss addresses that follows x, then for the windowed Markov predictor, we use $N'(x) = \{ P(y_i \mid x) \mid i \leq w, x, y_i \in$ **T** $\}$. For our experiments, we chose w to be five. Another important modification is that we do not necessarily use up

all p prefetch request slots. Of the p top probabilities of $N'(x)$, we discard those that are less than a threshold. The idea is to minimize bandwidth requirement by not prefetching those addresses with low probabilities.

4.3 Hidden Markov Model (HMM) Predictor

The Hidden Markov Model (HMM) is a well-known technique that has a wide range of applications [10, 16, 21]. Essentially, it is a Markov chain where each state generates an observation. HMM are known to be very useful for time-series modeling since the discrete state-space can be used to approximate many non-linear, non-Gaussian systems.

A HMM can be characterize as follows. Let S be the number of states, and K be the number of (unique) symbols. The model consists of three matrices:

- $A_{i,j}$ is the probability of making a transition from state i to state j, with the requirement that $\sum_j A_{i,j} = 1$;
- $B_{i,k}$ is the probability of outputting symbol k when in state i, with the requirement that $\sum_k B_{i,k} = 1$;
- π_i is the probability of starting in state I, with $\sum_i \pi_i = 1$

There are established algorithms to train a HMM. These include the Viterbi and Baum-Welch algorithms [6]. We used a modified version of a publicly available HMM code used for speech recognition [7] to create HMMs of a sample trace, **T**. A unique HMM is created for each hotspot. We set K to be the number of unique miss addresses in **T**. Each pass through a hotspot is taken to be a unique training sequence.

To obtain the prediction table from the trained HMM, we used the following strategy. Given $x \in \mathbf{T}$, we sort the set $\{B_{i,x} \mid i \in S\}$ and obtain the states $i_1, i_2, \ldots, i_k, \ldots, i_q$

corresponding to the highest q members of the sorted set. For each of these states, we sort the set $\{A_{i_k,j} \mid j \in S\}$ and obtain the r highest probability next state. For each of these next states, we again select the q highest probability from $\{B_{j,y} \mid j \in S, y \in K\}$. From this we can construct a length two sequence (x, y) as well as its associated probability $P(x, y)$ where

$$P(x, y) = B_{x,i_k} \times A_{i_k,j_l} \times B_{j_l,y}$$

Proceeding in a similar manner, we can construct sequences of any length together with their associated probabilities. In practice, for our experiments, we stopped at sequences of length 3 as the longer the sequence, the lower its associated probability. With all these sequences up to a certain length in hand, we sort them according to their probabilities. We then proceed to pick p unique symbols that are members of the sequences of highest probabilities as entries in the prediction table for x. To overcome the problem of two miss addresses mapping to the same prediction table location, the same technique outlined in section 4.1 is used.

5. Experimental Setup

We use the Trimaran compiler-EPIC architecture simulation infrastructure [24] to evaluate the performance of our proposed system and of each of the three off-line learning algorithms outlined above. We compared the performance of our system against that of using larger caches, and the RPT hardware prefetch scheme of Chen and Baer [5]. The following benchmarks were used for the evaluation:

- 052.alvin benchmark from SPEC 92 [23] which trains a neural network using back propagation.
- 130.li from SPEC 95 which is a Xlisp interpreter.

- 181.mcf from SPEC CPU2000 which does combinatorial optimization / single-depot vehicle scheduling
- 183.equake from SPECfp 2000 which does wave propagation simulation.
- bisort, mst, treeadd, health from Olden Pointer Benchmark suite.

Our baseline setup is an IA64-like EPIC machine [14] with four integer, two floating point and two memory units and a 32Kbyte L1 cache and a 256Kbyte L2 cache. We computed stall cycles for L1 and L2 load misses when L1 cache size is 32K, 64K and 128K with 256K L2 cache. In our experiments, the predictor is used to prefetch data from L2 into a 32Kbyte prefetch buffer co-located with the L1 cache.

Our main metric for characterizing the performance of the memory system is *stall cycles*. Stall cycles account for a significant portion of actual data intensive program run-time (up to 90% in some of modern architectures) and significant portion of stall cycles comes from load misses. Reduction in stall cycles therefore directly leads to performance improvement. Since our EPIC machine is an in-order machine, we assumed a "stall-upon-use" latency model. In this stalling model, a load instruction that causes a cache miss will not immediately block the pipeline. The pipeline is stall only at the earliest attempt to use the data that is to be loaded.

There are three parameters used to compute stall cycles. First is the *minimum def-use latency* which is the minimum number of cycles for a certain value to be used after it is loaded by a load instruction. This is obtained from the compiler. The second set of parameters consists of the miss penalties for load misses at the level one and the level two caches. In our experiments, a L1 cache load miss costs 7 cycles and a L2 cache load miss costs 32 cycles. Finally, the clock cycles at each L1 load miss occurred are also used.

The stall cycles for L1 load misses *without prediction* using profiling is computed as follows:

For certain load *X* operation,

- If *X* results in a L1 cache hit
  - Stall cycle += min (H - L, 0)
- If *X* results in a miss at L1 but a hit at L2
  - Stall cycle += min (M_1 - L, 0)

where H is the hit latency, L is minimum def-use latency and M_1 is miss penalty for L1 cache. The stall cycles for L1 load misses *with* our prediction is computed as follows:

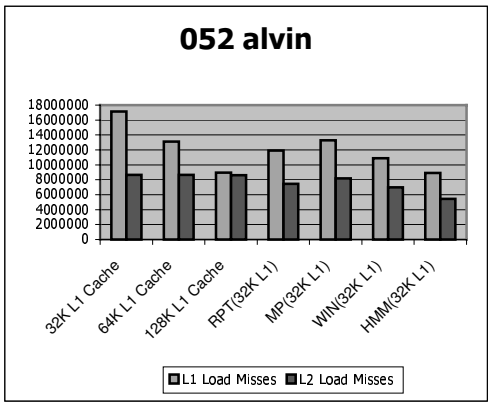
For certain load *X* operation that was correctly predicted

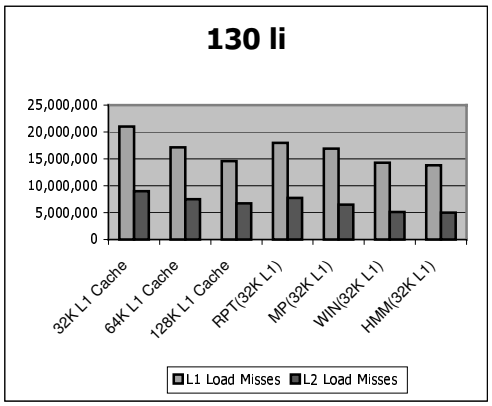
- And *X* results in a cache hit
  - Stall cycle += min (H - L, 0)
- And *X* results in a cache miss at L1 but a hit at L2
  - Stall cycle += min (M_1 - d - L, 0)

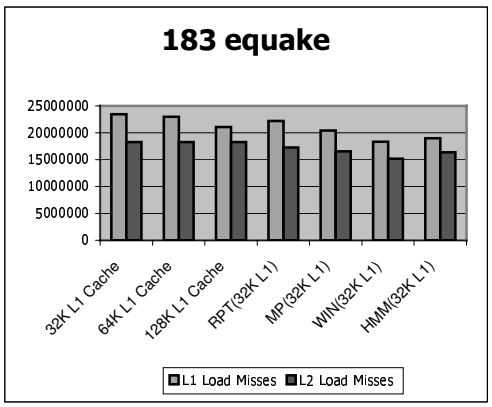
where d is distance in terms of clock cycles between load *X* and the previous request to prefetch *X*. If a load operation was not preceded by any prefetch request, then the computation of stall cycles is same as that without prediction. We should point out that we did not consider store misses as most load misses dominated in the benchmarks. Furthermore, the same traces used to train the predictors were used in the evaluation. (*Results from using different inputs were still not available at the time of submission*.) We shall now present the results of our experiments.

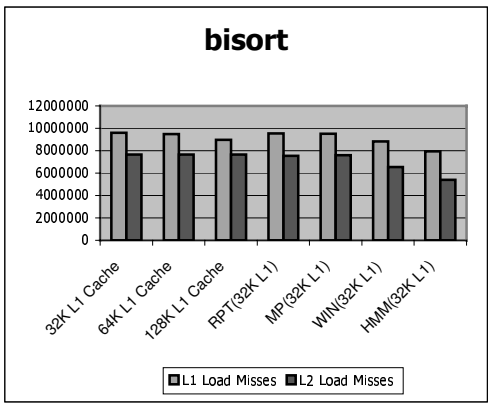
6. Results

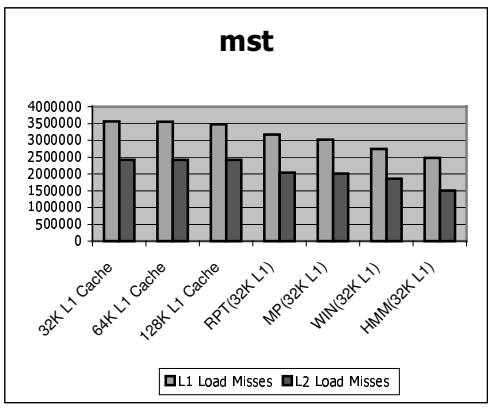
We measured how many load misses occurred during simulation (Fig. 2). The results shows that increasing L1 cache size does not necessarily improve performance especially for data intensive applications using dynamic data structures like pointers.

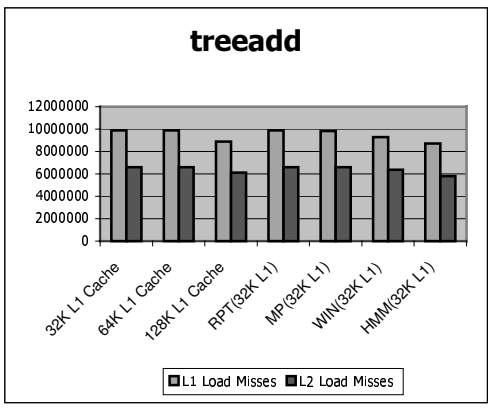

052 alvin
18000000
16000000
14000000
12000000
10000000
8000000
6000000
4000000
2000000
0
32K L1 Cache
64K L1 Cache
128K L1 Cache
RPT(32K L1)
MP(32K L1)
WIN(32K L1)
HMM(32K L1)
L1 Load Misses
L2 Load Misses

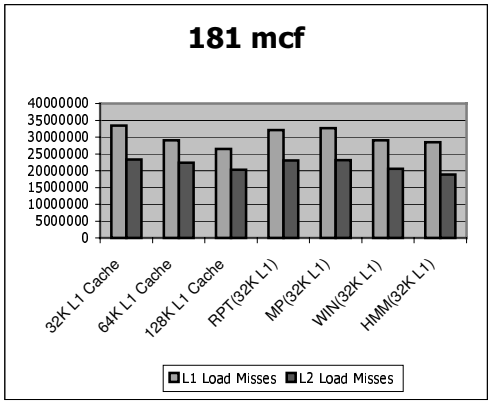

130 li
25,000,000
20,000,000
15,000,000
10,000,000
5,000,000
0
32K L1 Cache
64K L1 Cache
128K L1 Cache
RPT(32K L1)
MP(32K L1)
WIN(32K L1)
HMM(32K L1)
L1 Load Misses
L2 Load Misses


183 equake
25000000
20000000
15000000
10000000
5000000
0
32K L1 Cache
64K L1 Cache
128K L1 Cache
RPT(32K L1)
MP(32K L1)
WIN(32K L1)
HMM(32K L1)
L1 Load Misses
L2 Load Misses


bisort
12000000
10000000
8000000
6000000
4000000
2000000
0
32K L1 Cache
64K L1 Cache
128K L1 Cache
RPT(32K L1)
MP(32K L1)
WIN(32K L1)
HMM(32K L1)
L1 Load Misses
L2 Load Misses


mst
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
32K L1 Cache
64K L1 Cache
128K L1 Cache
RPT(32K L1)
MP(32K L1)
WIN(32K L1)
HMM(32K L1)
L1 Load Misses
L2 Load Misses


treeadd
12000000
10000000
8000000
6000000
4000000
2000000
0
32K L1 Cache
64K L1 Cache
128K L1 Cache
RPT(32K L1)
MP(32K L1)
WIN(32K L1)
HMM(32K L1)
L1 Load Misses
L2 Load Misses



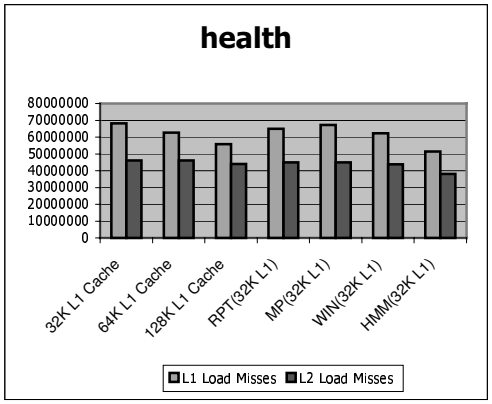


Fig. 2. Load misses with various L1 cache size, RPT, Markovian Predictor

Next we measure total dynamic cycles and stall cycles for L1 and L2 load misses. The results are shown in Fig.3.

	052 alvin	130 li	183 equake	bisort	mst	Tree-add	181 mcf	health
Total stall cycles with 32K L1 cache	376M	407M	714M	299M	96M	266M	919M	1,813M
Total stall cycles with 64K L1 cache	350M	338M	710M	298M	96M	266M	861M	1,778M
Total stall cycles with 128K L1 cache	320M	298M	694M	291M	96M	244M	790M	1,705M
Total stall cycles with RPT	305M	350M	670M	292M	83M	266M	912M	1,794M
Total stall cycles with Markov Predictor	335M	299M	637M	294M	81M	265M	917M	1,807M
Total stall cycles with Windowed Markov Predictor	284M	247M	582M	260M	74M	254M	806M	1,718M

Total stall cycles with Hidden Markov Predictor	224M	241M	622M	219M	62M	234M	751M	1,476M
Total compute cycles	57M	45M	61M	37M	23M	42M	92M	238M

Fig 3. Total stall cycles(L1 + L2 load miss cycles), and compute cycles (both in millions) for each benchmark

The percentage performance improvement is shown in normalized graph of Fig. 4 with the base case being that of a machine with 32KByte L1 cache and 256KByte L2 cache without using any prediction scheme. As can be seen, the Windowed Markov predictor showed a bigger performance increase in compared to bigger cache size or RPT or simple Markov Predictor scheme except one SPEC 2000 benchmmark(181mcf) and two olden pointer benchmarks(treeadd and health). The HMM Predictor (HMP) in turn did better than the Windowed Markov Predictor (WMP) or any other schemes in all benchmark tests except one SPEC 2000fp benchmark(183equake) where Windowed Markov Predictor was slightly better than Hidden Markov Predictor. In one instance, a 37% improvement in performance was recorded using Hidden Markov Predictor. In almost all cases, the use of off-line learning algorithms gave a pronounced performance improvement over that of simply increasing the cache size or a hardware prefetch scheme like RPT.

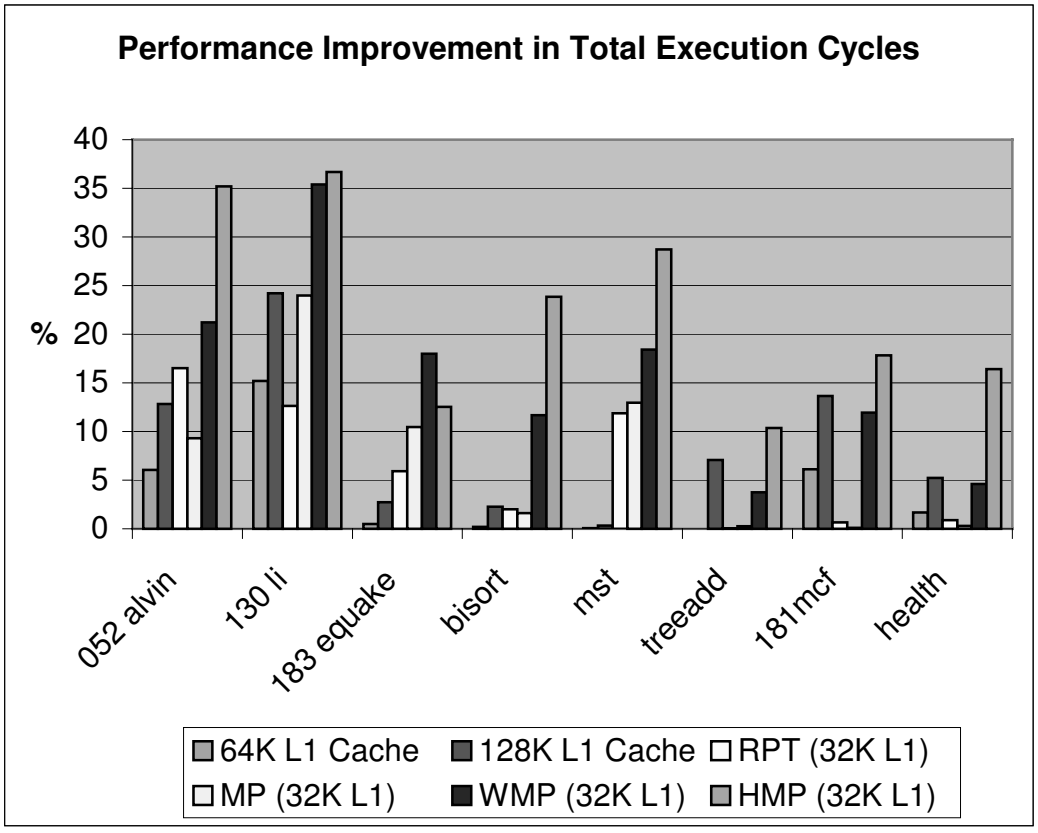


Fig 4. Performance improvement in total cycles by percentage wise (normalized by 32K L1 cache and 256K L2 cache without using any prefetching scheme).

7. Conclusion

In this paper, we proposed a paradigm and architectural framework for the use of off-line learning algorithms in the prefetching of data. In most cases, the use of off-line learning scheme gives a much better performance than merely increasing the size of the level 1 cache size. This trend will most likely appear as same in case of level 2 cache as well. With one exception of 183.equake, the hidden Markov model outperforms other implementations including Markov predictor with window size of 5. This result highlights the potential of adapting the hidden Markov model which is already popular in many

other fields to overcome the memory bottleneck problem. The increasing L1 cache size alone did not help much improving data cache performance and this trend is more significant as application shows more data intensive characteristics as seen most of the olden pointer benchmark suite. In those cases, using sophisticated off-line learning scheme coupled with the generic architecture we proposed, has more advantage over bigger caches. Our future research seeks to develop more powerful learning module with effective hardware supported prediction engine.

References

[1] J.–L. Baer and T.–F. Chen, "An effective on-chip preloading scheme to reduce data access penalty." In *Proceedings of Supercomputing '91,* Pages 176 – 186. 1991.

[2] D. Callahan, K. Kennedy and A. Porterfield, "Software Prefetching," In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 40 – 52, 1991.

[3] M.J. Charney and A.P. Reeves, "Generalized correlation based hardware prefetching." *Technical Report EE-CEG-95-1, Cornell University*, Feb 1995.

[4] T.-F. Chen and J.-L. Baer, "A performance study of software and hardware data prefetching schemes." In *Proceedings of the 21st Annual International Symposium on Computer Architecture,* pp 223 – 232. 1994.

[5] T.-F. Chen, J.–L. Baer, "Effective hardware-based data prefetching for high-performance processor Computers." *IEEE Transactions on Computers, Volume: 44-5,* pp. 609-623. May 1995.

[6] J.R. Deller, Jr., J.G. Proakis, and J.H.L. Hansen, *Discrete-time Processing of Speech Signals*. MacMillan 1993.

[7] Discrete HMM Toolkit. http://www.isip.msstate.edu/projects/speech/software/discrete_hmm/index.html

[8] J. W. C. Fu and J. H. Patel, “Data prefetching strategies for vector cache memories.” In *International Parallel Processing Symposium*, 1991.

[9] J. W. C. Fu and J. H. Patel, “Stride directed prefetching in scalar processors.” In In *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 102-110, 1992.

[10] F. Jelinek, “Self-organized language modeling for speech recognition.” *Technical report, IBM T. J. Watson Research Center,* 1985.

[11] D. Joseph, D. Grunwald, “Prefetching using Markov predictors.” In *Proceedings of the 24th Annual International Symposium on Computer Architecture,* pages 252-263, 1997.

[12] N. P. Jouppi, “Improving direct-mapped cache performance by the addition of a smally, fully associative cache and prefetch buffers,” In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 364-373, 1990.

[13] M. Karlsson, F. Dahlgren, and P. Stenstrom, “A prefetching technique for irregular accesses to linked data structures.” In *Proceedings of the Sixth International Sysmposium on High-Performance Computer Architecture.* pp. 206 –217. 2000.

[14] V. Kathail, M.S. Schlansker, and B.R. Rau, “HPL-PD Architectural Specifications: Version 1.1.” *Hewlett-Packard Laboratories Technical Report HPL-93-80(R.1).* Revised 2000. http://www.trimaran.org/docs/hpl-pd.pdf.

[15] A. C. Klaiber, and H. M. Levy, “An architecture for software-controlled data prefetching,” In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 43 – 53, 1991.

[16] A. Krogh, S.I. Mian and D. Haussler, “A hidden Markov model that finds genes in E. Coli DNA.” In *Nucleic Acids Research,* Vol. 22, No. 22, pp. 4769-4778, 1994.

[17] M. Lipasti, W. Schmidt, S. Kunkel, R. Roediger, “SPAID: Software prefetching in Pointer and Call-intensive environment.” In *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 231 - 236, 1995.

[18] C.-K. Luk and T.C. Mowry, “Compiler-based prefetching for recursive data structures.” *In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 222-233. 1996.

[19] S. Mehrota, and H. Luddy, “Examination of a memory classification scheme for pointer intensive and numeric programs.” *Technical Report CRSD Tech. Report 1351*, CRSD, University of Illinois, Dec 1995.

[20] T. C. Mowry, M. S. Lam and A. Gupta, “Design and Evaluation of a Compiler Algorithm for Prefetching.” In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 62-73, 1992.

[21] A. Nadas, “Estimation of probabilities in the language model of the IBM speech recognition system.” In *IEEE Transaction on Acoustics, Signal and Speech Processing,* 32(4): 859 - 861, 1984.

[22] T. Ozawa, Y. Kimura, and S. Nishizaki, “Cache miss heuristics and preloading techniques for general-purpose programs.” In *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 243 - 248, 1995.

[23] The SPEC benchmarks. http://www.spec.org

[24] The Trimaran Compiler Infrastructure. http://www.trimaran.org

[25] S.P. Vanderwiel, and D.J. Lilja, “A compiler-assisted data prefetch controller.” In *Proceedings of International Conference on Computer Design,* pp. 372 –377. 1999.

[26] S.P. Vanderwiel and D. J. Lilja, “Data Prefetch Mechanisms”, *ACM Computing Survey*, vol. 32, no. 2, pp. 174 – 199. Jun 2000.